Graph II

Jason
Prerequisite

• Graph 1
  • Definition of graphs
  • Graph representations
  • DFS and BFS
Table of Contents

• Trees
• Directed Acyclic Graph
• Other Special Graphs
• More on DFS and BFS
Cycles

• Cause trouble in graphs
• Make problems difficult
• If there are no cycles, problems can be solved efficiently
Trees

• Either one of the followings is the definition
  • A connected graph with $|V|-1$ edges
  • A connected graph without cycles
  • A graph with exactly one path between every pair of vertices
Rooted Tree

• Sometimes the edges can be directional
• One vertex has been designated the root
• The edges are directional and all away from the root
Terms on directed tree

- Parent
- Sibling
- Children
Terms on directed tree

Ancestors
Terms on directed tree

Descendants
Terms on directed tree

Subtrees
Terms on directed tree
Terms on directed tree

Height = 4
Terms on directed tree

Depth

0
1
2
3
4
Tree Implementation

- Trees are also graphs
- Use graph representations
  - Adjacency matrix
  - Adjacency lists
  - Edge list

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Trees - Usage

• Some problems are reduced or become easier
• Special algorithms can be used
Trees - Usage

• Shortest Path?
  • “A graph with exactly one path between every pair of vertices”

• Minimum Spanning Tree?????
  • It is a tree
Trees - Usage

• Can also be used as data structures by storing data in a specific way
  • Binary Search Trees
  • Heaps
  • Tries
  • Segment Trees
  • Suffix Trees
  • ...
Binary Tree

- A rooted tree which all vertices (nodes) have at most 2 children
Perfect Binary Tree

• A binary tree in which all non-leaf nodes have two children and all leaves have the same depth
Complete Binary Tree

- A perfect binary tree with some or all rightmost leaf nodes removed
Complete Binary Tree

• A complete binary tree of height $h$ has between $2^h$ and $2^{h+1} - 1$ nodes
• The height of a complete binary tree with $n$ nodes is $\lfloor \log n \rfloor$
• Instead of using pointers or lists, it can be implemented using arrays
Binary Tree Implementation

• Use pointers
• Each pointer stores one of the node’s children

```c
struct node{
    int value;
    node* left, right;
}
```

• May be difficult to trace
• Useful when implementing balanced BST (e.g. AVL Tree)
Complete Binary Tree Implementation

• The root node is stored in array position 1
• For any element in array position i:
  • The left child is in position 2 * i
  • The right child is in the cell after the left child (2 * i + 1)
  • The parent is in position ⌊i / 2⌋
Complete Binary Tree Implementation

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[]</td>
<td>B</td>
<td>C</td>
<td>F</td>
<td>A</td>
<td>G</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: A complete binary tree with nodes A, B, C, D, E, F, and G.
Complete Binary Tree Implementation

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[]</td>
<td>B</td>
<td>C</td>
<td>F</td>
<td>A</td>
<td>G</td>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Depth = 0
Depth = 1
Depth = 2
Complete Binary Tree Implementation

• Nodes with depth $h$ are stored in position $2^h$ to $2^{h+1}-1$

• Space Complexity : $O(N)$

• Useful in implementing tree data structures
  • Binary Heap
  • Segment tree
Tree Traversal

- Like graphs, nodes in a tree can also be visited using Depth First Search (DFS) and Breath First Search (BFS)
Depth First Search

- The search tree is deepened as much as possible on each child before going to the next sibling

procedure DFS(vertex v){
  label v as visited
  for all w in children of v do
    DFS(w)
}
Depth First Search

• Usually we only want to process every node exactly once
• For a binary tree, we have three orders for tree traversal:
  • Pre-order traversal
  • In-order traversal
  • Post-order traversal
Depth First Search – Pre-order

• Starting from the root
• For the traversal at node N:
  • Process the node N
  • Recursively traverse its left subtree
  • Recursively traverse its right subtree
Depth First Search – Pre-order
Depth First Search – Pre-order

A B C D E F G H
Depth First Search – Pre-order

A | B | D
---|---|---

A
---
B
---
D
---
E
---
G
---
H
---
C
---
F
Depth First Search – Pre-order
Depth First Search – Pre-order

A | B | D | E | G |
---|---|---|---|---|
A  | B  | D  | E  | G  |

Diagram:

- A (root)
  - B
    - D
    - E
    - G
  - C
    - F
  - H
Depth First Search – Pre-order

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
</table>

A → B → D → E → G → H
Depth First Search – Pre-order

A  B  D  E  G  H  C

A — B — D — E — G — H — C

A — C — F

A — B — D — E — G — H — C

A — B — D — E — G — H — F
Depth First Search – Pre-order
Depth First Search – Pre-order
Depth First Search – In-order

• Starting from the root
• For the traversal at node N:
  • Recursively traverse its left subtree
  • Process the node N
  • Recursively traverse its right subtree
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – In-order

D | B | G | E

A → B → D → G → E → H → F

A → C → F
Depth First Search – In-order
Depth First Search – In-order

D  B  G  E  H  A

A
↓
B
↓
D  E
↓
G  H
↓
C  F
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – In-order
Depth First Search – Post-order

• Starting from the root
• For the traversal at node N:
  • Recursively traverse its left subtree
  • Recursively traverse its right subtree
  • Process the node N
Depth First Search – Post-order
Depth First Search – Post-order
Depth First Search – Post-order
Depth First Search – Post-order
Depth First Search – Post-order

D  G

D  G

Depth First Search (DFS) is a search algorithm for traversing or searching tree or graph data structures. Depth First Search visits vertices of a graph in a depthward motion and uses backtracking to search further down paths than those already traversed. In the context of a tree structure, post-order traversal visits the children of a node before visiting the node itself. The diagram illustrates a tree with nodes labeled A, B, C, D, E, F, G, H, and D, G. The tree is traversed in a depth-first manner, and the post-order traversal sequence is implied by the order in which the nodes are visited: D, G, E, B, H, C, F, A.
Depth First Search – Post-order

D  G  H

A

B

D  G  H

C

E

F

H
Depth First Search – Post-order
Depth First Search – Post-order

D G H E B
Depth First Search – Post-order
Depth First Search – Post-order

A → B → D → G → E → H → C → F
Depth First Search – Post-order
Depth First Search – Post-order
Depth First Search – Post-order
Depth First Search

- Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

<table>
<thead>
<tr>
<th>Pre-order</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order</td>
<td>D</td>
<td>B</td>
<td>G</td>
<td>E</td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>F</td>
</tr>
</tbody>
</table>
Depth First Search

• Given either pre-order or post-order paired with in-order, the structure of the binary tree can be known

<table>
<thead>
<tr>
<th>Pre-order</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order</td>
<td>D</td>
<td>B</td>
<td>G</td>
<td>E</td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>F</td>
</tr>
</tbody>
</table>
Breadth First Search

- Starts at the root
- Explores all children first before moving to the next level
Breadth First Search

Procedure bfs()
    push the root into the queue
    while the queue is not empty do
        t = front of the queue
        process node t
        for all w in children of t do
            push w into the queue
        pop t from the queue
    }
Example 1

• Given a rooted tree, find the number of descendants of each node
Example 1

• To find the number of descendants of node N, find the number of descendants of all of its children first
• The answer will be the sum of the answer of its children + number of its children
• Depth First Search using Post-order
Example 2

- Given a tree, find the diameter of the tree, i.e. the longest distance in the tree
Example 2

- Although the root is not specified, we can randomly choose a node to be the root
Example 2

• For each vertex, find the longest and second-longest distance from its descendants
• Children’s answers are used
• Depth First Search
Example 2

• Or we can find the deepest node from the root first (using DFS)
Example 2

• Or we can find the deepest node from the root first (using DFS)
• Then find the deepest node from it
More usage

• See Dynamic Programming II
Directed Acyclic Graph

- Other than trees, some directed graph may have no cycles as well
- We call them Directed Acyclic Graphs (DAG)
Directed Acyclic Graph

- We may want to process the nodes layer by layer
- Nodes having edges to others should be manipulated first
Topological Order

• An ordering of vertices in a directed acyclic graph, such that if there is a path from $v_i$ to $v_j$, then $v_j$ appears after $v_i$ in the ordering.

• For example, one of the topological orders of this graph is $A D G E H B I F C$
Topological Sort

• Find a vertex with no incoming edges. Number it and remove all outgoing edges from it
• Repeat the above process
• Can be implemented by DFS or BFS
Topological Sort

• Define the indegree of a vertex \( v \) as the number of edges connecting to \( v \)

• Instead of removing the vertices and edges, we can manipulate the indegree of all vertices

• When a vertex is removed, lower the indegree of the nodes connected by it by 1
Topological Sort

• Topological order:

```
1 2 4 3 6 5 7 0
```
Topological Sort

• Topological order:

```
1 2 4 3 6 5 7
```
Topological Sort

- Topological order: [1, 2]
Topological Sort

• Topological order:

```
1 2 4
```

```
3 5 6 7
```
Topological Sort

- Topological order:

```
    1  2  4  3
```
Topological Sort

- Topological order:

```
1 2 4 3 5
```
Topological Sort

• Topological order:

```
1 2 4 3 5 6
```
Topological Sort

• Topological order: 1 2 4 3 5 6 7
Topological Sort

Procedure Tsort(vertex v){
    put v into the order
    visited[v] = 1
    for all edge from v to u do
        indegree[u] = indegree[u] – 1
        if indegree[u] = 0 and visited[u] = 0 then
            Tsort(u)
}

For all vertex v do
    if visited[v] = 0 then
        Tsort(v)
Topological Sort

• Time Complexity: \( O(|V|^2) \) or \( O(|V|+|E|) \) (same as DFS and BFS)

• Using the topological order, we can process the nodes correctly

• All information from previous layers can be retrieved by next layer
Example

- Given a DAG, find the shortest path from one node to the other
Example

• Find the topological order first: AFIDGBHEC
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Example

• Find the topological order first: AFIDGBHEC
• Calculate the shortest paths based on the topological order
Detecting cycles

- Topological sort can find an order in a DAG
- What will happen if the graph is not acyclic?
Detecting cycles
Detecting cycles
Detecting cycles

STUCK
Detecting cycles

• If the topological sort can not be finished, there are cycles in the graph
Other Special Graphs

• Beside of trees and DAG, there are other kinds of graphs that can make problems easier
Chain

- All vertices have 2 neighbors, except two of them only have 1 neighbor
Chain

- No cycles
- No branches
- Easy calculations
- Process the nodes from head to tail
Star

- A tree with one internal node and k leaves
Star

• No cycles
• Maximum distance = 2
• Easy calculations
• Special handle the internal node
Unicyclic Graph

• A connected graph with exactly one cycle
Unicyclic Graph

- $|V| = |E|$
- Work on the cycle first
- Process the subtrees connected to the cycle one by one
Complete Graph

• Every pair of distinct vertices is connected by a unique edge
• $|E| = |V| \times (|V| - 1) / 2$
Planar Graph

• A graph whose vertices and edges can be drawn in a plane such that no two of the edges intersect
• Using its properties, more efficient algorithms can be applied
Bipartite Graph

• A graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V

• That is, you can assign one of two colours for each node such that all edges have different colour on two side
Bipartite Graph

• A graph whose vertices can be divided into two disjoint and independent sets U and V such that every edge connects a vertex in U to one in V

• That is, you can assign one of two colours for each node such that all edges have different colour on two side
Bipartite Graph

- Some graphs may be special properties that guarantees it is bipartite
  - Trees
  - Rectangular Grid
  - Or something like this:
Bipartite Graph

- An odd cycle cannot be bicolored
- A graph is bipartite if and only if it does not contain an odd cycle
- To check whether a graph is bipartite, we can check if the graph contains cycles with odd length
Bipartite Graph

- Perform DFS/BFS on any node
- Assign 0 and 1 alternatively to every node according to the depth
  - If a node is assigned 0, assign 1 to its neighbors, and vice versa
- If any edge has same number on two sides, the graph is not bipartite
Bipartite Graph

• A bipartite graph can also reduce the difficulty of many problems
  • Independent Set
  • Vertex Cover
  • Edge Cover
  • Matching
  • (won’t cover here)
More on DFS and BFS

• As mentioned in Graph 1, DFS cannot find shortest paths in a normal unweighted graph
• However, we can modify it
Iterative deepening depth-first search

• A depth-limited version of depth-first search
• Run repeatedly with increasing depth limits until the goal is found
Iterative deepening depth-first search

• A depth-limited version of depth-first search
• Run repeatedly with increasing depth limits until the goal is found
• Limit = 1
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 2
Iterative deepening depth-first search

• A depth-limited version of depth-first search
• Run repeatedly with increasing depth limits until the goal is found
• Limit = 3
Iterative deepening depth-first search

- A depth-limited version of depth-first search
- Run repeatedly with increasing depth limits until the goal is found
- Limit = 4
Iterative deepening depth-first search

Procedure ids(){
    limit = 0
    while goal is not found
        limit = limit + 1
        dfs(root,limit)
}

Procedure dfs(node, depth){
    if depth<0 return
    process the node
    for all neighbor w of node
        if w is not visited
            dfs(w, depth – 1)
}
Iterative deepening depth-first search

• Same as Breadth First Search
• Much less memory is used
Bidirectional search

- Sometimes BFS is not fast enough
- Too many useless nodes/states are visited
Bidirectional search

• Runs two searches simultaneously
• One forward from the initial state
• One backward from the goal
• Stop when the two meet in the middle
Bidirectional search
Bidirectional search
Bidirectional search
Bidirectional search
Bidirectional search

Procedure bds{
    push starting point and ending point into the queue
    while the queue is not empty do
        t = front of the queue
        for all vertex w adjacent to t do
            if w is not visited
                calculate the distance to w from the original direction
                push w into the queue
            else if w was visited from the other direction
                calculate the distance to w
                return the distance from starting point to w + the distance from w to ending point
        pop t from the queue
}
Bidirectional search

• The running time is better than BFS
• Less irrelevant searching is done
• Useful in state searching